

MMIC FILE COPY

(2)

AVF Control Number: AVF-VSR-130.0988
87-11-17-VRX

DTIC
ELECTE
FEB 13 1989
D^{cs}
AD-A204 129

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 880201W1.09020
Verdix Corporation
VAda-010-3131, Version 5.5
Prime EXL

Completion of On-Site Testing:
5 FEBRUARY 1988

Prepared By:
Ada Validation Facility
ASD/SCSL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

89 2 13 126

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	12. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Verdix Corporation, VAda-010-3131, Version 5.5, Prime EXL (Host and Target). (880201W1.09020)		5. TYPE OF REPORT & PERIOD COVERED 5 Feb 1988 to 5 Feb 1989
7. AUTHOR(s) Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A.		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A.		8. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A.		12. REPORT DATE 5 Feb 1988
		13. NUMBER OF PAGES 37 p.
		15. SECURITY CLASS (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report) UNCLASSIFIED		
18. SUPPLEMENTARY NOTES		
19. KEYWORDS (Continue on reverse side if necessary and identify by block number) Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) VAda-010-3131, Version 5.5, Verdix Corporation, Wright-Patterson Air Force Base, Prime EXL under UNIX System V, Release 3.0 (Host and Target), ACVC 1.9.		

DD FORM 1473

1 JAN 73

EDITION OF 1 NOV 85 IS OBSOLETE

S/N 0102-LF-014-8601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Accepted for filing
NHS
DUE
H. J. ...
J
A-1

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	PURPOSE OF THIS VALIDATION SUMMARY REPORT	1-2
1.2	USE OF THIS VALIDATION SUMMARY REPORT	1-2
1.3	REFERENCES	1-3
1.4	DEFINITION OF TERMS	1-3
1.5	ACVC TEST CLASSES	1-4
CHAPTER 2	CONFIGURATION INFORMATION	
2.1	CONFIGURATION TESTED	2-1
2.2	IMPLEMENTATION CHARACTERISTICS	2-2
CHAPTER 3	TEST INFORMATION	
3.1	TEST RESULTS	3-1
3.2	SUMMARY OF TEST RESULTS BY CLASS	3-1
3.3	SUMMARY OF TEST RESULTS BY CHAPTER	3-2
3.4	WITHDRAWN TESTS	3-2
3.5	INAPPLICABLE TESTS	3-2
3.6	TEST, PROCESSING, AND EVALUATION MODIFICATIONS	3-3
3.7	ADDITIONAL TESTING INFORMATION	3-4
3.7.1	Prevalidation	3-4
3.7.2	Test Method	3-4
3.7.3	Test Site	3-5
APPENDIX A	CONFORMANCE STATEMENT	
APPENDIX B	APPENDIX F OF THE Ada STANDARD	
APPENDIX C	TEST PARAMETERS	
APPENDIX D	WITHDRAWN TESTS	

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- . To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- . To attempt to identify any unsupported language constructs required by the Ada Standard
- . To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc., under the direction of the AVF according to policies and procedures established by the Ada Validation Organization (AVO). On-site testing was conducted from 1 February 1988 through 5 February 1988 at Aloha, OR.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

INTRODUCTION

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983.
2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established procedures.
AVO	The Ada Validation Organization. In the context of this report, the AVO is responsible for establishing procedures for compiler validations.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

INTRODUCTION

Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Language Maintenance Panel	The Language Maintenance Panel (LMP) is a committee established by the Ada Board to recommend interpretations and possible changes to the ANSI/MIL-STD for Ada.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	An Ada program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. However, no checks are performed during execution to see if the test objective has been met. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or

semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of these units is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an

INTRODUCTION

illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2
CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: VAda-010-3131, Version 5.5

ACVC Version: 1.9

Certificate Number: 880201W1.09020

Host Computer:

Machine:	Prime EXL
Operating System:	UNIX System V, Release 3.0
Memory Size:	11 megabytes

Target Computer:

Machine:	Prime EXL
Operating System:	UNIX System V, Release 3.0
Memory Size:	11 megabytes

CONFIGURATION INFORMATION

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- . Capacities.

The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- . Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed `SYSTEM.MAX_INT`. This implementation processes 64-bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- . Predefined types.

This implementation supports the additional predefined types `SHORT_INTEGER`, `SHORT_FLOAT`, and `TINY_INTEGER` in the package `STANDARD`. (See tests B86001C and B86001D.)

- . Based literals.

An implementation is allowed to reject a based literal with a value exceeding `SYSTEM.MAX_INT` during compilation, or it may raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` during execution. This implementation raises `NUMERIC_ERROR` during execution. (See test E24101A.)

- . Expression evaluation.

Apparently no default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

CONFIGURATION INFORMATION

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Apparently `NUMERIC_ERROR` is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Sometimes `NUMERIC_ERROR` is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is gradual. (See tests C45524A..Z.)

. Rounding.

The method used for rounding to integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round to even. (See test C4A014A.)

. Array types.

An implementation is allowed to raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` for an array having a `'LENGTH` that exceeds `STANDARD.INTEGER'LAST` and/or `SYSTEM.MAX_INT`. For this implementation:

Declaration of an array type or subtype declaration with more than `SYSTEM.MAX_INT` components raises no exception. (See test C36003A.)

`NUMERIC_ERROR` is raised when `'LENGTH` is applied to an array type with `INTEGER'LAST + 2` components. (See test C36202A.)

`NUMERIC_ERROR` is raised when `'LENGTH` is applied to an array type with `SYSTEM.MAX_INT + 2` components. (See test C36202B.)

A packed `BOOLEAN` array having a `'LENGTH` exceeding `INTEGER'LAST` raises `NUMERIC_ERROR` when the array type is declared. (See test C52103X.)

A packed two-dimensional `BOOLEAN` array with more than `INTEGER'LAST` components raises `NUMERIC_ERROR` when the array subtype is declared. (See test C52104Y.)

CONFIGURATION INFORMATION

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with discriminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

- Representation clauses.

The Ada Standard does not require an implementation to support representation clauses. If a representation clause is not supported, then the implementation must reject it.

CONFIGURATION INFORMATION

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

Record representation clauses are not supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

. Pragma.

The pragma `INLINE` is supported for functions and procedures. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

. Input/output.

The package `SEQUENTIAL_IO` can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package `DIRECT_IO` can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

Modes `IN_FILE` and `OUT_FILE` are supported for `SEQUENTIAL_IO`. (See tests CE2102D and CE2102E.)

Modes `IN_FILE`, `OUT_FILE`, and `INOUT_FILE` are supported for `DIRECT_IO`. (See tests CE2102F, CE2102I, and CE2102J.)

CONFIGURATION INFORMATION

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to last element written. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can be created in OUT_FILE mode, and can be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for SEQUENTIAL_IO, TEXT_IO, and DIRECT_IO for both reading and writing. (See tests CE2107A..I (9 tests), CE2110B, CE2111D, CE2111H, CE3111A..E (5 tests), CE3114B, and CE3115A.)

An external file associated with more than one internal file can be deleted for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.)

Temporary files are given names and deleted when closed. (See tests CE2108A and CE2108C.)

. Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3
TEST INFORMATION

3.1 TEST RESULTS

At the time of testing, Version 1.9 of the ACVC comprised 3122 tests of which 24 had been withdrawn. Of the remaining tests, 226 were determined to be inapplicable to this implementation. Not all of the inapplicable tests were processed during testing; 201 executable tests that use floating-point precision exceeding that supported by the implementation were not processed. Modifications to the code, processing, or grading for 24 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	109	1049	1633	17	18	46	2872
Failed	0	0	0	0	0	0	0
Inapplicable	1	2	223	0	0	0	226
Withdrawn	3	2	18	0	1	0	24
TOTAL	113	1053	1874	17	19	46	3122

TEST INFORMATION

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER														TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14		
Passed	190	500	541	245	166	98	142	326	137	36	234	3	254	2872	
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Inapplicable	14	73	134	3	0	0	1	1	0	0	0	0	0	226	
Withdrawn	2	13	2	0	0	1	2	0	0	0	2	1	1	24	
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122	

3.4 WITHDRAWN TESTS

The following 24 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C
C35904A	C35A03E	C35A03R	C37213H	C37213J
C37215C	C37215E	C37215G	C37215H	C38102C
C41402A	C45614C	A74106C	C85018B	C87B04B
CC1311B	BC3105A	AD1A01A	CE2401H	

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 226 tests were inapplicable for the reasons indicated:

- . C35702B uses LONG_FLOAT which is not supported by this implementation.
- . A39005G uses a record representation clause which is not supported by this compiler.

TEST INFORMATION

- . The following tests use LONG_INTEGER, which is not supported by this compiler.

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

- . C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.
- . C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.
- . C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.
- . C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.
- . The following 201 tests require a floating-point accuracy that exceeds the maximum of 15 digits supported by this implementation:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made with the approval of the AVO, and are made in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into sub-tests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 24 Class B tests.

TEST INFORMATION

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B24009A	B24204A..C	B2A003A..C	B33301A	B37201A
B38003A	B38003B	B38009A	B38009B	B41202A
B44001A	B64001A	B67001A..D	B91001H	B91003B
B95001A	BC1303F	BC3005B		

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the VAdA-010-3131 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the VAdA-010-3131 compiler using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a Prime EXL operating under UNIX System V, Release 3.0.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded onto a MicroVAX II operating under VMS. The test files were transferred via FTP and Ethernet to the Prime EXL computer used for testing.

After the test files were loaded to disk, the full set of tests was compiled, linked, and executed (as appropriate) on the Prime EXL. Results were printed and examined.

The compiler was tested using command scripts provided by Verdix Corporation and reviewed by the validation team. The compiler was tested using all switch settings in their default positions.

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

TEST INFORMATION

3.7.3 Test Site

Testing was completed on 5 February 1988.

APPENDIX A

CONFORMANCE STATEMENT

Verdix Corporation has submitted the following
conformance statement concerning the VAda-010-3131
compiler.

DECLARATION OF CONFORMANCE

Compiler Implementer: Verdix Corporation
Ada Validation Facility: ASD/SCEL Wright Patterson AFB, OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration


Base Compiler Name:	VAda-010-3131	Version: 5.5
Host Architecture	ISA: Prime EXL	OS&VER #: Unix System V rel 3.0
Target Architecture	ISA: Prime EXL	OS&VER #: Unix System V rel 3.0

Derived Compiler Registration

Derived Compiler Name:	VAda-010-3131	Version: 5.5
Host Architecture	ISA: Intel System 301	OS&VER #: Unix System V rel 3.0
Target Architecture	ISA: Intel System 301	OS&VER #: Unix System V rel 3.0
Host Architecture	ISA: Intel System 320	OS&VER #: Unix System V rel 3.0
Target Architecture	ISA: Intel System 320	OS&VER #: Unix System V rel 3.0
Host Architecture	ISA: AT&T 6386	OS&VER #: Unix System V rel 3.0
Target Architecture	ISA: AT&T 6386	OS&VER #: Unix System V rel 3.0

Implementer's Declaration

I, the undersigned, representing Verdix Corporation, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Verdix Corporation is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.




Verdix Corporation
Gregory Burns, Project Manager

Date: 2/5/85

Owner's Declaration

I, the undersigned, representing Verdix Corporation take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.



Verdix Corporation
Gregory Burns, Project Manager

Date: 2/5/88

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of MIL-STD-1815A, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the VAda-010-3131, Version 5.5, are described in the following sections which discuss topics in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A). Implementation-specific portions of the package STANDARD are also included in this appendix.

package STANDARD is

...

type INTEGER is range -2_147_483_648 .. 2_147_483_647;

type SHORT_INTEGER is range -32768 .. 32767;

type TINY_INTEGER is range -128 .. 127;

type FLOAT is digits 15 range -1.798E+308 .. 1.798E+308;

type SHORT_FLOAT is digits 6 range -3.403E+38 .. 3.403E+38;

type DURATION is delta 1.000E-03 range -2147483.648 .. 2147483.647;

...

end STANDARD;

APPENDIX F. Implementation-Dependent Characteristics

1. Implementation-Dependent Pragmas

1.1. `INLINE_ONLY` Pragma

The `INLINE_ONLY` pragma, when used in the same way as pragma `INLINE`, indicates to the compiler that the subprogram must *always* be inlined. This pragma also suppresses the generation of a callable version of the routine which save code space.

1.2. `BUILT_IN` Pragma

The `BUILT_IN` pragma is used in the implementation of some predefined Ada packages, but provides no user access. It is used only to implement code bodies for which no actual Ada body can be provided, for example the `MACHINE_CODE` package.

1.3. `SHARE_CODE` Pragma

The `SHARE_CODE` pragma takes the name of a generic instantiation or a generic unit as the first argument and one of the identifiers `TRUE` or `FALSE` as the second argument. This pragma is only allowed immediately at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

When the first argument is a generic unit the pragma applies to all instantiations of that generic. When the first argument is the name of a generic instantiation the pragma applies only to the specified instantiation, or overloaded instantiations.

If the second argument is `TRUE` the compiler will try to share code generated for a generic instantiation with code generated for other instantiations of the same generic. When the second argument is `FALSE` each instantiation will get a unique copy of the generated code. The extent to which code is shared between instantiations depends on this pragma and the kind of generic formal parameters declared for the generic unit.

The name pragma `SHARE_BODY` is also recognized by the implementation and has the same effect as `SHARE_CODE`. It is included for comparability with earlier versions of VADS.

1.4. `NO_IMAGE` Pragma

The pragma suppresses the generation of the image array used for the `IMAGE` attribute of enumeration types. This eliminates the overhead required to store the array in the executable image.

1.5. `EXTERNAL_NAME` Pragma

The `EXTERNAL_NAME` pragma takes the name of a subprogram or variable defined in Ada and allows the user to specify a different external name that may be used to reference the entity from other languages. The pragma is allowed at the place of : declarative item in a package specification and must apply to an object declared earlier in the same package specification.

1.6. `INTERFACE_OBJECT` Pragma

The `INTERFACE_OBJECT` pragma takes the name of a variable defined in another language and allows it to be referenced directly in Ada. The pragma will replace all occurrences of the variable name with an external reference to the second, `link_argument`. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification. The object must be declared as a scalar or an access type. The object *cannot* be

any of the following:

- a loop variable,
- a constant,
- an initialized variable,
- an array, or
- a record.

1.7. IMPLICIT_CODE Pragma

Takes one of the identifiers ON or OFF as the single argument. This pragma is only allowed within a machine code procedure. It specifies that implicit code generated by the compiler be allowed or disallowed. A warning is issued if OFF is used and any implicit code needs to be generated. The default is ON.

2. Implementation of Predefined Pragmas

2.1. CONTROLLED

This pragma is recognized by the implementation but has no effect.

2.2. ELABORATE

This pragma is implemented as described in Appendix B of the Ada RM.

2.3. INLINE

This pragma is implemented as described in Appendix B of the Ada RM.

2.4. INTERFACE

This pragma supports calls to 'C' and FORTRAN functions. The Ada subprograms can be either functions or procedures. The types of parameters and the result type for functions must be scalar, access or the predefined type ADDRESS in SYSTEM. An optional third argument overrides the default link name. All parameters must have mode IN. Record and array objects can be passed by reference using the ADDRESS attribute.

2.5. LIST

This pragma is implemented as described in Appendix B of the Ada RM.

2.6. MEMORY_SIZE

This pragma is recognized by the implementation but has no effect. The implementation does not allow SYSTEM to be modified by means of pragmas; the SYSTEM package must be recompiled.

2.7. OPTIMIZE

This pragma is recognized by the implementation but has no effect.

2.8. PACK

This pragma will cause the compiler to choose a non-aligned representation for composite types. It will not cause objects to be packed at the bit level.

2.9. PAGE

This pragma is implemented as described in Appendix B of the Ada RM.

2.10. PRIORITY

This pragma is implemented as described in Appendix B of the Ada RM.

2.11. SHARED

This pragma is recognized by the implementation but has no effect.

2.12. STORAGE_UNIT

This pragma is recognized by the implementation but has no effect. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

2.13. SUPPRESS

This pragma is implemented as described, except that RANGE_CHECK and DIVISION_CHECK cannot be suppressed.

2.14. SYSTEM_NAME

This pragma is recognized by the implementation but has no effect. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

3. Implementation-Dependent Attributes

3.1. P'REF

For a prefix that denotes an object, a program unit, a label, or an entry:

This attribute denotes the effective address of the first of the storage units allocated to P. For a subprogram, package, task unit, or label, it refers to the address of the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, it refers to the corresponding hardware interrupt. The attribute is of the type OPERAND defined in the package MACHINE_CODE. The attribute is only allowed within a machine code procedure.

(For a package, task unit, or entry, the 'REF attribute is not supported.)

4. Specification Of Package SYSTEM

```

package SYSTEM
is
    type NAME is ( balance_dynix );

    SYSTEM_NAME          : constant NAME := balance_dynix;

    STORAGE_UNIT         : constant := 8;
    MEMORY_SIZE          : constant := 16_777_216;

    -- System-Dependent Named Numbers

    MIN_INT              : constant := -2_147_483_647 - 1;
    MAX_INT              : constant := 2_147_483_647;
    MAX_DIGITS           : constant := 15;
    MAX_MANTISSA         : constant := 31;
    FINE_DELTA           : constant := 2.0**(-31);
    TICK                 : constant := 0.01;

    -- Other System-dependent Declarations

    subtype PRIORITY is INTEGER range 0 .. 99;

    MAX_RBC_SIZE : integer := 64*1024;

    type ADDRESS is private;

    NO_ADDR: constant ADDRESS;

    function PHYSICAL_ADDRESS(I: INTEGER) return ADDRESS;
    function ADDR_GT(A, B: ADDRESS) return BOOLEAN;
    function ADDR_LT(A, B: ADDRESS) return BOOLEAN;
    function ADDR_GE(A, B: ADDRESS) return BOOLEAN;
    function ADDR_LE(A, B: ADDRESS) return BOOLEAN;
    function ADDR_DIFF(A, B: ADDRESS) return INTEGER;
    function INCR_ADDR(A: ADDRESS; INCR: INTEGER) return ADDRESS;
    function DECR_ADDR(A: ADDRESS; DECR: INTEGER) return ADDRESS;

    function ">"(A, B: ADDRESS) return BOOLEAN renames ADDR_GT;
    function "<"(A, B: ADDRESS) return BOOLEAN renames ADDR_LT;
    function ">=" (A, B: ADDRESS) return BOOLEAN renames ADDR_GE;
    function "<=" (A, B: ADDRESS) return BOOLEAN renames ADDR_LE;
    function "-"(A, B: ADDRESS) return INTEGER renames ADDR_DIFF;
    function "+"(A: ADDRESS; INCR: INTEGER) return ADDRESS renames INCR_ADDR;
    function "-"(A: ADDRESS; DECR: INTEGER) return ADDRESS renames DECR_ADDR;

    pragma inline(PHYSICAL_ADDRESS);
    pragma inline(ADDR_GT);
    pragma inline(ADDR_LT);
    pragma inline(ADDR_GE);
    pragma inline(ADDR_LE);
    pragma inline(ADDR_DIFF);
    pragma inline(INCR_ADDR);
    pragma inline(DECR_ADDR);

private
    type ADDRESS is new INTEGER;
    no_addr: constant address := 0;

end SYSTEM;

```

5. Restrictions On Representation Clauses

5.1. Pragma PACK

Bit packing is not supported. Objects and larger components are packed to the nearest whole STORAGE_UNIT.

5.2. Size Specification

The size specification T'SMALL is not supported except when the representation specification is the same as the value 'SMALL for the base type.

5.3. Record Representation Clauses

Components clauses must be aligned on STORAGE_UNIT boundaries.

5.4. Address Clauses

Address clauses are supported for variables and constants.

5.5. Interrupts

Interrupt entries are supported for UNIX signals. The Ada for clause gives the UNIX signal number.

5.6. Representation Attributes

The ADDRESS attribute is not supported for the following entities:

- Packages
- Tasks
- Labels
- Entries

5.7. Machine Code Insertions

Machine code insertions are supported.

The general definition of the package MACHINE_CODE provides an assembly language interface for the target machine. It provides the necessary record type(s) needed in the code statement, an enumeration type of all the opcode mnemonics, a set of register definitions, and a set of addressing mode functions.

The general syntax of a machine code statement is as follows:

```
CODE_n'( opcode, operand {, operand} );
```

where *n* indicates the number of operands in the aggregate.

A special case arises for a variable number of operands. The operands are listed within a subaggregate. The format is as follows:

```
CODE_N'( opcode, (operand {, operand}) );
```

For those opcodes that require no operands, named notation must be used (cf. RM 4.3(4)).

```
CODE_0'( op => opcode );
```

The *opcode* must be an enumeration literal (i.e. it cannot be an object, attribute, or a rename).

An *operand* can only be an entity defined in MACHINE_CODE or the 'REF attribute.

The arguments to any of the functions defined in MACHINE_CODE must be static expressions, string literals, or the functions defined in MACHINE_CODE. The 'REF attribute may not be used as an argument in any of these functions.

Inline expansion of machine code procedures is supported.

6. Conventions for Implementation-generated Names

There are no implementation-generated names.

7. Interpretation of Expressions in Address Clauses

Address clauses are supported for constants and variables. Interrupt entries are specified with the number of the UNIX signal.

8. Restrictions on Unchecked Conversions

None.

9. Restrictions on Unchecked Deallocations

None.

10. Implementation Characteristics of I/O Packages

Instantiations of `DIRECT_IO` use the value `MAX_REC_SIZE` as the record size (expressed in `STORAGE_UNITS`) when the size of `ELEMENT_TYPE` exceeds that value. For example for unconstrained arrays such as string where `ELEMENT_TYPE'SIZE` is very large, `MAX_REC_SIZE` is used instead. `MAX_RECORD_SIZE` is defined in `SYSTEM` and can be changed by a program before instantiating `DIRECT_IO` to provide an upper limit on the record size. In any case the maximum size supported is $1024 \times 1024 \times \text{STORAGE_UNIT}$ bits. `DIRECT_IO` will raise `USE_ERROR` if `MAX_REC_SIZE` exceeds this absolute limit.

Instantiations of `SEQUENTIAL_IO` use the value `MAX_REC_SIZE` as the record size (expressed in `STORAGE_UNITS`) when the size of `ELEMENT_TYPE` exceeds that value. For example for unconstrained arrays such as string where `ELEMENT_TYPE'SIZE` is very large, `MAX_REC_SIZE` is used instead. `MAX_RECORD_SIZE` is defined in `SYSTEM` and can be changed by a program before instantiating `INTEGER_IO` to provide an upper limit on the record size. `SEQUENTIAL_IO` imposes no limit on `MAX_REC_SIZE`.

11. Implementation Limits

The following limits are actually enforced by the implementation. It is not intended to imply that resources up to or even near these limits are available to every program.

11.1. Line Length

The implementation supports a maximum line length of 500 characters including the end of line character.

11.2. Record and Array Sizes

The maximum size of a statically sized array type is $4,000,000 \times \text{STORAGE_UNITS}$. The maximum size of a statically sized record type is $4,000,000 \times \text{STORAGE_UNITS}$. A record type or array type declaration that exceeds these limits will generate a warning message.

11.3. Default Stack Size for Tasks

In the absence of an explicit `STORAGE_SIZE` length specification every task except the main program is allocated a fixed size stack of 10,240 `STORAGE_UNITS`. This is the value returned by `T'STORAGE_SIZE` for a task type `T`.

11.4. Default Collection Size

In the absence of an explicit `STORAGE_SIZE` length attribute the default collection size for an access type is 100,000 `STORAGE_UNITS`. This is the value returned by `T STORAGE_SIZE` for an access type `T`.

11.5. Limit on Declared Objects

There is an absolute limit of 6,000,000 x `STORAGE_UNITS` for objects declared statically within a compilation unit. If this value is exceeded the compiler will terminate the compilation of the unit with a **FATAL** error message.

APPENDIX C TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

<u>Name and Meaning</u>	<u>Value</u>
\$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	1..498 => 'A', 499 => '1'
\$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	1..498 => 'A', 499 => '2'
\$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	1..249 => 'A', 250 => '3', 251..499 => 'A'
\$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	1..249 => 'A', 250 => '4', 251..499 => 'A'
\$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	1..496 => '0', 497..499 => "298"

TEST PARAMETERS

Name and Meaning	Value
\$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	1..493 => '0', 494..499 => "69.0E1"
\$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1..249 => 'A')
\$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1..249 => 'A', 250 => '1')
\$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..479 => ' ')
\$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_647
\$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2_147_483_647
\$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	"/illegal/file_name/2{]}\$%2102C.DAT"
\$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	"/illegal/file_name/CE2102C*.DAT"
\$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0

<u>Name and Meaning</u>	<u>Value</u>
\$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10_000_000.0
\$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	"/no/such/directory/" & "ILLEGAL_EXTERNAL_FILE_NAME1"
\$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	"/no/such/directory/" & "ILLEGAL_EXTERNAL_FILE_NAME2"
\$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2_147_483_648
\$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2_147_483_647
\$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2_147_483_648
\$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
\$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-10_000_000.0
\$MAX_DIGITS Maximum digits supported for floating-point types.	15
\$MAX_IN_LEN Maximum input line length permitted by the implementation.	499
\$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2_147_483_647
\$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648

TEST PARAMETERS

<u>Name and Meaning</u>	<u>Value</u>
<p>\$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.</p>	<p>(1..2 => "2#", 3..496 => '0', 497..499 => "11#")</p>
<p>\$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.</p>	<p>(1..3 => "16:", 4..495 => '0', 496..499 => "F.E:")</p>
<p>\$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.</p>	<p>(1..499 => 'A')</p>
<p>\$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.</p>	<p>-2_147_483_648</p>
<p>\$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.</p>	<p>TINY_INTEGER</p>
<p>\$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.</p>	<p>16#FFFFFFFFD#</p>

APPENDIX D
WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 24 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- . B28003A: A basic declaration (line 36) incorrectly follows a later declaration.
- . E28005C: This test requires that "PRAGMA LIST (ON);" not appear in a listing that has been suspended by a previous "PRAGMA LIST (OFF);". The Ada Standard is not clear on this point, and the matter will be reviewed by the AJPO.
- . C34004A: The expression in line 168 yields a value outside the range of the target type T, but there is no handler for CONSTRAINT_ERROR.
- . C35502P: The equality operators in lines 62 and 69 should be inequality operators.
- . A35902C: The assignment in line 17 of the nominal upper bound of a fixed-point type to an object raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.
- . C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.
- . C35A03E and C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.
- . C37213H: The subtype declaration of SCONS in line 100 is incorrectly expected to raise an exception when elaborated.

WITHDRAWN TESTS

- . C37213J: The aggregate in line 451 incorrectly raises `CONSTRAINT_ERROR`.
- . C37215C, C37215E, C37215G, and C37215H: Various discriminant constraints are incorrectly expected to be incompatible with type `CONS`.
- . C38102C: The fixed-point conversion on line 23 wrongly raises `CONSTRAINT_ERROR`.
- . C41402A: The attribute `'STORAGE_SIZE` is incorrectly applied to an object of an access type.
- . C45614C: The function call of `IDENT_INT` in line 15 uses an argument of the wrong type.
- . A74106C, C85018B, C87B04B, and CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising `CONSTRAINT_ERROR`. Errors of this sort occur at lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively.
- . BC3105A: Lines 159 through 168 expect error messages, but these lines are correct Ada.
- . AD1A01A: The declaration of subtype `SINT3` raises `CONSTRAINT_ERROR` for implementations which select `INT'SIZE` to be 16 or greater.
- . CE2401H: The record aggregates in lines 105 and 117 contain the wrong values.